FOUNDATION-0.4: Qdrant Vector Database - PART 1 (Code)

🎯 CONTEXT

Phase: FOUNDATION (Week 1 - Day 2 Evening)

Component: Qdrant Vector Database Setup

Estimated Time: 15 min AI execution + 10 min verification

Complexity: MEDIUM

Risk Level: LOW

Files: Part 1 of 2 (Code implementation)

MILESTONE: Vector storage for agent memory and learning! 🧠

📦 DEPENDENCIES

Must Complete First:

- **P-01:** Bootstrap Project Structure ✅ COMPLETED
- **FOUNDATION-0.2a-0.2e:** PostgreSQL schemas ✅ COMPLETED
- **FOUNDATION-0.3:** ClickHouse ✅ COMPLETED

Required Services Running:

```bash
# Verify all services are healthy
cd ~/optiinfra
make verify

# Expected output:
# PostgreSQL... ✅ HEALTHY
# ClickHouse... ✅ HEALTHY
# Qdrant... ✅ HEALTHY
# Redis... ✅ HEALTHY
```

🎯 OBJECTIVE

Set up **Qdrant vector database** to enable agents to learn from past decisions and retrieve relevant context.

What We're Building:

3 Vector Collections:

1. **cost_optimization_knowledge** - Past cost optimization decisions and outcomes
2. **performance_patterns** - Successful performance optimization patterns
3. **customer_context** - Customer-specific learning and preferences

Python Client:

- Easy-to-use interface for embeddings
- Similarity search for context retrieval
- Outcome tracking for learning

Why Qdrant?

Feature	Traditional DB	Qdrant
Semantic search	❌	✅
Similarity matching	❌	✅
Vector storage	❌	✅ Optimized
Speed	N/A	Sub-millisecond
Best for	Exact match	Semantic similarity

Architecture:

```
┌─────────────────────────────────────────┐
│         POSTGRESQL (0.2a-0.2e)          │
│    (Structured transactional data)      │
└────────────────┬────────────────────────┘
                 │
                 │ IDs & References
                 ▼
┌─────────────────────────────────────────┐
│        QDRANT (0.4) ✅ NEW               │
│     (Vector semantic memory)            │
│                                         │
│  ┌───────────────────────────────────┐  │
│  │ cost_optimization_knowledge       │  │
│  │ - Past decisions                  │  │
│  │ - Outcomes (saved/wasted)         │  │
│  │ - Context (when it worked)        │  │
│  └───────────────────────────────────┘  │
```

```
│                              │
│   ┌─────────────────────────────────────┐   │
│   │ performance_patterns              │   │
│   │ - Successful optimizations        │   │
│   │ - Configuration that worked       │   │
│   │ - Similar past scenarios          │   │
│   └─────────────────────────────────────┘   │
│                              │
│   ┌─────────────────────────────────────┐   │
│   │ customer_context                  │   │
│   │ - Customer preferences            │   │
│   │ - Past interactions               │   │
│   │ - Specific constraints            │   │
│   └─────────────────────────────────────┘   │
└─────────────────────────────────────────────┘
```

Use Cases:

Cost Agent:

> "I want to migrate to spot instances. What worked for similar customers?" → Searches `cost_optimization_knowledge` for similar migrations → Returns: "3 similar cases, 2 succeeded (35% savings), 1 failed (workload too variable)"

Performance Agent:

> "Customer has high P95 latency. What optimizations worked before?" → Searches `performance_patterns` for similar scenarios → Returns: "KV cache tuning worked in 5 similar cases (2.3x improvement)"

Application Agent:

> "Customer prefers slow rollouts. What's their risk tolerance?" → Searches `customer_context` for preferences → Returns: "Customer X prefers 10%→25%→50%→100% rollout with 24hr validation"

📁 FILE 1: Qdrant Collection Schemas

Location: `~/optiinfra/shared/qdrant/schemas/collections.py`

```python
```

```python
"""
Qdrant collection schemas and configurations.

Defines the structure and settings for all vector collections
used by OptiInfra agents for memory and learning.
"""

from dataclasses import dataclass
from typing import Dict, Any
from qdrant_client.models import Distance, VectorParams


@dataclass
class CollectionConfig:
    """Configuration for a Qdrant collection."""

    name: str
    vector_size: int
    distance: Distance
    description: str
    payload_schema: Dict[str, str]

    def to_vector_params(self) -> VectorParams:
        """Convert to Qdrant VectorParams."""
        return VectorParams(
            size=self.vector_size,
            distance=self.distance
        )


# ============================================================================
# COLLECTION CONFIGURATIONS
# ============================================================================

COST_OPTIMIZATION_KNOWLEDGE = CollectionConfig(
    name="cost_optimization_knowledge",
    vector_size=1536,  # OpenAI ada-002 embedding size
    distance=Distance.COSINE,
    description="Knowledge base of past cost optimization decisions and outcomes",
    payload_schema={
        "optimization_id": "UUID - Link to optimizations table",
        "customer_id": "UUID - Which customer",
        "optimization_type": "String - spot_migration, reserved_instance, right_sizing",
```

```python
        "decision_context": "String - Why this decision was made",
        "outcome": "String - success, failed, rolled_back",
        "savings_percent": "Float - Actual savings achieved (if success)",
        "cost_impact": "Float - Dollar savings per month",
        "execution_date": "DateTime - When this was executed",
        "cloud_provider": "String - aws, gcp, azure",
        "instance_type": "String - m5.xlarge, etc",
        "workload_characteristics": "String - Description of workload",
        "lessons_learned": "String - What we learned from this",
        "confidence_score": "Float - How confident we were (0-1)",
        "customer_feedback": "String - Customer's feedback (if any)"
    }
)

PERFORMANCE_PATTERNS = CollectionConfig(
    name="performance_patterns",
    vector_size=1536,
    distance=Distance.COSINE,
    description="Successful performance optimization patterns",
    payload_schema={
        "optimization_id": "UUID - Link to optimizations table",
        "customer_id": "UUID - Which customer",
        "service_type": "String - vllm, tgi, sglang",
        "model_name": "String - Which LLM model",
        "optimization_applied": "String - What was changed",
        "problem_description": "String - Original performance issue",
        "solution_description": "String - How it was solved",
        "before_latency_p95": "Float - P95 latency before (ms)",
        "after_latency_p95": "Float - P95 latency after (ms)",
        "improvement_factor": "Float - How much faster (2.3x, etc)",
        "config_changes": "JSONB - Specific configuration changes",
        "side_effects": "String - Any negative impacts observed",
        "execution_date": "DateTime - When applied",
        "stability_score": "Float - How stable post-change (0-1)",
        "replicable": "Boolean - Can this be repeated for similar cases"
    }
)

CUSTOMER_CONTEXT = CollectionConfig(
    name="customer_context",
    vector_size=1536,
    distance=Distance.COSINE,
    description="Customer-specific context, preferences, and constraints",
    payload_schema={
```

```
        "customer_id": "UUID - Which customer",
        "context_type": "String - preference, constraint, historical_note",
        "topic": "String - What this context is about",
        "content": "String - The actual context/note",
        "source": "String - How we learned this (conversation, observation, explicit)",
        "confidence": "Float - How confident we are (0-1)",
        "created_at": "DateTime - When this context was added",
        "updated_at": "DateTime - Last update",
        "priority": "String - low, medium, high, critical",
        "applies_to_agents": "List[String] - Which agents should use this",
        "examples": "String - Example scenarios where this applies",
        "exceptions": "String - When this doesn't apply"
    }
)


# ============================================================================
# COLLECTION REGISTRY
# ============================================================================

ALL_COLLECTIONS = [
    COST_OPTIMIZATION_KNOWLEDGE,
    PERFORMANCE_PATTERNS,
    CUSTOMER_CONTEXT
]


def get_collection_config(collection_name: str) -> CollectionConfig:
    """
    Get configuration for a specific collection.

    Args:
        collection_name: Name of the collection

    Returns:
        CollectionConfig for the requested collection

    Raises:
        ValueError: If collection not found
    """
    for config in ALL_COLLECTIONS:
        if config.name == collection_name:
            return config
```

```python
raise ValueError(f"Unknown collection: {collection_name}")
```

📁 FILE 2: Qdrant Client

Location: `~/optiinfra/shared/qdrant/client.py`

python

```python
"""
Qdrant client for vector storage and similarity search.

Provides easy-to-use interface for:
- Storing optimization decisions with embeddings
- Searching for similar past decisions
- Learning from outcomes

Usage:
    from shared.qdrant.client import get_qdrant_client

    client = get_qdrant_client()

    # Store a decision
    client.store_cost_decision(
        optimization_id="...",
        decision_context="Migrating to spot for stable batch workload",
        outcome="success",
        savings_percent=38.5,
        ...
    )

    # Search for similar decisions
    results = client.search_similar_cost_decisions(
        query="migrate to spot instances for batch processing",
        limit=5
    )
"""

from qdrant_client import QdrantClient
from qdrant_client.models import (
    Distance, VectorParams, PointStruct, Filter,
    FieldCondition, MatchValue, SearchRequest
)
from typing import List, Dict, Any, Optional
import os
import uuid
from datetime import datetime
import logging

from shared.qdrant.schemas.collections import (
    ALL_COLLECTIONS,
    COST_OPTIMIZATION_KNOWLEDGE,
```

```python
    PERFORMANCE_PATTERNS,
    CUSTOMER_CONTEXT,
    get_collection_config
)

logger = logging.getLogger(__name__)


class QdrantVectorClient:
    """Client for vector storage and similarity search in Qdrant."""

    def __init__(self):
        """Initialize Qdrant client."""
        self.client = QdrantClient(
            host=os.getenv('QDRANT_HOST', 'localhost'),
            port=int(os.getenv('QDRANT_PORT', 6333)),
            api_key=os.getenv('QDRANT_API_KEY', None)
        )

        # Embedding function (you'll integrate OpenAI/other provider)
        self.embedding_function = self._default_embedding_function

        logger.info(f"Qdrant client initialized: {self.client._client.rest_uri}")

    def _default_embedding_function(self, text: str) -> List[float]:
        """
        Default embedding function (placeholder).

        In production, replace with actual embedding model:
        - OpenAI ada-002
        - Sentence-Transformers
        - Custom model

        Args:
            text: Text to embed

        Returns:
            List of floats (embedding vector)
        """
        # TODO: Replace with actual embedding model
        # For now, return random vector for testing
        import random
        return [random.random() for _ in range(1536)]
```

```python
    def ping(self) -> bool:
        """
        Check if Qdrant is accessible.

        Returns:
            bool: True if Qdrant responds, False otherwise
        """
        try:
            collections = self.client.get_collections()
            return True
        except Exception as e:
            logger.error(f"Qdrant ping failed: {e}")
            return False

    def initialize_collections(self):
        """
        Create all collections if they don't exist.

        This is idempotent - safe to call multiple times.
        """
        existing = {c.name for c in self.client.get_collections().collections}

        for config in ALL_COLLECTIONS:
            if config.name not in existing:
                logger.info(f"Creating collection: {config.name}")
                self.client.create_collection(
                    collection_name=config.name,
                    vectors_config=config.to_vector_params()
                )
                logger.info(f"✅ Created collection: {config.name}")
            else:
                logger.info(f"Collection already exists: {config.name}")

    # ==========================================================================
    # COST OPTIMIZATION KNOWLEDGE
    # ==========================================================================

    def store_cost_decision(
        self,
        optimization_id: str,
        customer_id: str,
        optimization_type: str,
        decision_context: str,
        outcome: str,
```

```python
    savings_percent: Optional[float] = None,
    cost_impact: Optional[float] = None,
    **kwargs
) -> str:
    """
    Store a cost optimization decision with embedding.

    Args:
        optimization_id: UUID from optimizations table
        customer_id: UUID from customers table
        optimization_type: spot_migration, reserved_instance, right_sizing
        decision_context: Why this decision was made (text for embedding)
        outcome: success, failed, rolled_back
        savings_percent: Actual savings (if success)
        cost_impact: Dollar savings per month
        **kwargs: Additional payload fields

    Returns:
        str: Point ID in Qdrant

    Example:
        point_id = client.store_cost_decision(
            optimization_id="123e4567-e89b-12d3-a456-426614174000",
            customer_id="789e0123-e89b-12d3-a456-426614174000",
            optimization_type="spot_migration",
            decision_context="Migrating batch processing workload to spot instances. Workload is tolerant to interruptions and
            outcome="success",
            savings_percent=38.5,
            cost_impact=18000,
            cloud_provider="aws",
            instance_type="m5.xlarge",
            workload_characteristics="Batch ETL jobs, 4-6 hour runtime, can checkpoint",
            lessons_learned="Spot worked well for this workload. No interruptions in first month."
        )
    """
    # Generate embedding from decision context
    embedding = self.embedding_function(decision_context)

    # Create payload
    payload = {
        "optimization_id": optimization_id,
        "customer_id": customer_id,
        "optimization_type": optimization_type,
        "decision_context": decision_context,
```

```
            "outcome": outcome,
            "savings_percent": savings_percent,
            "cost_impact": cost_impact,
            "execution_date": datetime.now().isoformat(),
            **kwargs
        }

        # Generate point ID
        point_id = str(uuid.uuid4())

        # Store in Qdrant
        self.client.upsert(
            collection_name=COST_OPTIMIZATION_KNOWLEDGE.name,
            points=[
                PointStruct(
                    id=point_id,
                    vector=embedding,
                    payload=payload
                )
            ]
        )

        logger.info(f"Stored cost decision: {point_id}")
        return point_id

    def search_similar_cost_decisions(
        self,
        query: str,
        limit: int = 5,
        filter_outcome: Optional[str] = None,
        filter_type: Optional[str] = None
    ) -> List[Dict[str, Any]]:
        """
        Search for similar cost optimization decisions.

        Args:
            query: Natural language query describing the scenario
            limit: Number of results to return
            filter_outcome: Optional filter (success, failed, rolled_back)
            filter_type: Optional filter by optimization type

        Returns:
            List of similar decisions with scores
```

```
Example:
    results = client.search_similar_cost_decisions(
        query="migrate batch processing to spot instances",
        limit=5,
        filter_outcome="success"
    )

    for result in results:
        print(f"Score: {result['score']:.3f}")
        print(f"Context: {result['payload']['decision_context']}")
        print(f"Outcome: {result['payload']['outcome']}")
        print(f"Savings: {result['payload']['savings_percent']}%")
"""
# Generate query embedding
query_embedding = self.embedding_function(query)

# Build filter
must_conditions = []
if filter_outcome:
    must_conditions.append(
        FieldCondition(
            key="outcome",
            match=MatchValue(value=filter_outcome)
        )
    )
if filter_type:
    must_conditions.append(
        FieldCondition(
            key="optimization_type",
            match=MatchValue(value=filter_type)
        )
    )

search_filter = Filter(must=must_conditions) if must_conditions else None

# Search
results = self.client.search(
    collection_name=COST_OPTIMIZATION_KNOWLEDGE.name,
    query_vector=query_embedding,
    limit=limit,
    query_filter=search_filter
)

return [
```

```python
        {
            "id": result.id,
            "score": result.score,
            "payload": result.payload
        }
        for result in results
    ]

# ===========================================================================
# PERFORMANCE PATTERNS
# ===========================================================================

def store_performance_pattern(
    self,
    optimization_id: str,
    customer_id: str,
    service_type: str,
    model_name: str,
    problem_description: str,
    solution_description: str,
    before_latency_p95: float,
    after_latency_p95: float,
    **kwargs
) -> str:
    """
    Store a successful performance optimization pattern.

    Args:
        optimization_id: UUID from optimizations table
        customer_id: UUID from customers table
        service_type: vllm, tgi, sglang
        model_name: Which LLM model
        problem_description: Original issue (for embedding)
        solution_description: How it was solved (for embedding)
        before_latency_p95: P95 latency before (ms)
        after_latency_p95: P95 latency after (ms)
        **kwargs: Additional payload fields

    Returns:
        str: Point ID in Qdrant
    """
    # Combine problem + solution for embedding
    combined_text = f"{problem_description}\n\nSolution: {solution_description}"
    embedding = self.embedding_function(combined_text)
```

```python
        # Calculate improvement
        improvement_factor = before_latency_p95 / after_latency_p95 if after_latency_p95 > 0 else 0

        # Create payload
        payload = {
            "optimization_id": optimization_id,
            "customer_id": customer_id,
            "service_type": service_type,
            "model_name": model_name,
            "problem_description": problem_description,
            "solution_description": solution_description,
            "before_latency_p95": before_latency_p95,
            "after_latency_p95": after_latency_p95,
            "improvement_factor": improvement_factor,
            "execution_date": datetime.now().isoformat(),
            **kwargs
        }

        point_id = str(uuid.uuid4())

        self.client.upsert(
            collection_name=PERFORMANCE_PATTERNS.name,
            points=[
                PointStruct(
                    id=point_id,
                    vector=embedding,
                    payload=payload
                )
            ]
        )

        logger.info(f"Stored performance pattern: {point_id}")
        return point_id

    def search_similar_performance_patterns(
        self,
        query: str,
        limit: int = 5,
        filter_service_type: Optional[str] = None
    ) -> List[Dict[str, Any]]:
        """
        Search for similar performance optimization patterns.
```

```
        Args:
            query: Description of the performance issue
            limit: Number of results
            filter_service_type: Optional filter (vllm, tgi, sglang)

        Returns:
            List of similar patterns with scores
        """
        query_embedding = self.embedding_function(query)

        search_filter = None
        if filter_service_type:
            search_filter = Filter(
                must=[
                    FieldCondition(
                        key="service_type",
                        match=MatchValue(value=filter_service_type)
                    )
                ]
            )

        results = self.client.search(
            collection_name=PERFORMANCE_PATTERNS.name,
            query_vector=query_embedding,
            limit=limit,
            query_filter=search_filter
        )

        return [
            {
                "id": result.id,
                "score": result.score,
                "payload": result.payload
            }
            for result in results
        ]

    # ==========================================================================
    # CUSTOMER CONTEXT
    # ==========================================================================

    def store_customer_context(
        self,
        customer_id: str,
```

```python
    context_type: str,
    topic: str,
    content: str,
    confidence: float = 0.8,
    **kwargs
) -> str:
    """
    Store customer-specific context or preference.

    Args:
        customer_id: UUID from customers table
        context_type: preference, constraint, historical_note
        topic: What this context is about
        content: The actual context (for embedding)
        confidence: How confident we are (0-1)
        **kwargs: Additional payload fields

    Returns:
        str: Point ID in Qdrant

    Example:
        client.store_customer_context(
            customer_id="123e4567-e89b-12d3-a456-426614174000",
            context_type="preference",
            topic="rollout_strategy",
            content="Customer prefers slow, cautious rollouts with 24-hour validation periods between stages. They value stabi
            confidence=0.9,
            source="conversation with CTO on 2025-01-15",
            priority="high",
            applies_to_agents=["performance_agent", "cost_agent"]
        )
    """
    embedding = self.embedding_function(content)

    payload = {
        "customer_id": customer_id,
        "context_type": context_type,
        "topic": topic,
        "content": content,
        "confidence": confidence,
        "created_at": datetime.now().isoformat(),
        "updated_at": datetime.now().isoformat(),
        **kwargs
    }
```

```
        point_id = str(uuid.uuid4())

        self.client.upsert(
            collection_name=CUSTOMER_CONTEXT.name,
            points=[
                PointStruct(
                    id=point_id,
                    vector=embedding,
                    payload=payload
                )
            ]
        )

        logger.info(f"Stored customer context: {point_id}")
        return point_id

    def search_customer_context(
        self,
        customer_id: str,
        query: str,
        limit: int = 3
    ) -> List[Dict[str, Any]]:
        """
        Search for relevant customer context.

        Args:
            customer_id: Which customer
            query: What to search for
            limit: Number of results

        Returns:
            List of relevant context with scores
        """
        query_embedding = self.embedding_function(query)

        search_filter = Filter(
            must=[
                FieldCondition(
                    key="customer_id",
                    match=MatchValue(value=customer_id)
                )
            ]
        )
```

```
        results = self.client.search(
            collection_name=CUSTOMER_CONTEXT.name,
            query_vector=query_embedding,
            limit=limit,
            query_filter=search_filter
        )

        return [
            {
                "id": result.id,
                "score": result.score,
                "payload": result.payload
            }
            for result in results
        ]


# ======================================================================
# SINGLETON PATTERN
# ======================================================================

_qdrant_client = None

def get_qdrant_client() -> QdrantVectorClient:
    """
    Get singleton Qdrant client instance.

    Returns:
        QdrantVectorClient: Singleton client instance

    Example:
        client = get_qdrant_client()
        if client.ping():
            print("Qdrant is ready!")
    """
    global _qdrant_client
    if _qdrant_client is None:
        _qdrant_client = QdrantVectorClient()
    return _qdrant_client
```

📁 FILE 3: Package Initialization

Location: `~/optiinfra/shared/qdrant/__init__.py`

```python
```

```python
"""
Qdrant vector database package.

Provides vector storage and semantic search for:
- Cost optimization knowledge (past decisions → outcomes)
- Performance patterns (successful optimizations)
- Customer context (preferences, constraints)

Usage:
    from shared.qdrant import get_qdrant_client

    client = get_qdrant_client()
    client.initialize_collections()

    # Store decision
    client.store_cost_decision(...)

    # Search for similar
    results = client.search_similar_cost_decisions(...)
"""

from shared.qdrant.client import (
    QdrantVectorClient,
    get_qdrant_client
)

from shared.qdrant.schemas.collections import (
    COST_OPTIMIZATION_KNOWLEDGE,
    PERFORMANCE_PATTERNS,
    CUSTOMER_CONTEXT,
    ALL_COLLECTIONS,
    get_collection_config
)

__all__ = [
    'QdrantVectorClient',
    'get_qdrant_client',
    'COST_OPTIMIZATION_KNOWLEDGE',
    'PERFORMANCE_PATTERNS',
    'CUSTOMER_CONTEXT',
    'ALL_COLLECTIONS',
```

```
    'get_collection_config'
]
```

📁 FILE 4: Schema Package Init

Location: `~/optiinfra/shared/qdrant/schemas/__init__.py`

```python
"""
Qdrant schema definitions.
"""

from shared.qdrant.schemas.collections import (
    CollectionConfig,
    COST_OPTIMIZATION_KNOWLEDGE,
    PERFORMANCE_PATTERNS,
    CUSTOMER_CONTEXT,
    ALL_COLLECTIONS,
    get_collection_config
)

__all__ = [
    'CollectionConfig',
    'COST_OPTIMIZATION_KNOWLEDGE',
    'PERFORMANCE_PATTERNS',
    'CUSTOMER_CONTEXT',
    'ALL_COLLECTIONS',
    'get_collection_config'
]
```

📁 FILE 5: Update Requirements

Location: `~/optiinfra/shared/requirements.txt`

```txt
```

```
# Existing dependencies...
sqlalchemy==2.0.23
alembic==1.12.1
psycopg2-binary==2.9.9
clickhouse-driver==0.2.6

# Qdrant client (ADD THIS)
qdrant-client==1.7.0

# Other dependencies...
```

📁 FILE 6: README Documentation

Location: `~/optiinfra/shared/qdrant/README.md`

```markdown
```

Qdrant Vector Database

Vector storage and semantic search for OptiInfra agent memory and learning.

Overview

Qdrant enables agents to:
- **Learn from past decisions** - "What worked for similar scenarios?"
- **Retrieve relevant context** - "What do we know about this customer?"
- **Semantic search** - Find similar situations, not just exact matches

Architecture

Collections
1. **cost_optimization_knowledge** - Past cost optimization decisions
2. **performance_patterns** - Successful performance optimizations
3. **customer_context** - Customer-specific preferences and constraints

Each collection stores:
- **Vector embeddings** (1536-dim from OpenAI ada-002)
- **Metadata payload** (structured data about the decision/pattern)
- **Cosine similarity** for semantic matching

Usage

Initialize Collections
```bash
# Run once to create all collections
python << 'EOF'
from shared.qdrant import get_qdrant_client

client = get_qdrant_client()
client.initialize_collections()
print("✅ Collections initialized")
EOF
```

Python Client
```python
from shared.qdrant import get_qdrant_client

# Get client
client = get_qdrant_client()
```

```
# Check connection
if client.ping():
    print("✅ Qdrant connected!")

# Store a cost optimization decision
point_id = client.store_cost_decision(
    optimization_id="123e4567-e89b-12d3-a456-426614174000",
    customer_id="789e0123-e89b-12d3-a456-426614174000",
    optimization_type="spot_migration",
    decision_context="Migrating batch ETL workload to spot instances. Workload can handle interruptions with checkpointing
    outcome="success",
    savings_percent=38.5,
    cost_impact=18000,
    cloud_provider="aws",
    instance_type="m5.xlarge"
)

# Search for similar decisions
results = client.search_similar_cost_decisions(
    query="migrate batch processing to spot",
    limit=5,
    filter_outcome="success"
)

for result in results:
    print(
```